Reasoning about time-critical reactive systems: A case-study

André M. Rodrigues da Silva

DI - CCTC, Universidade do Minho
Campus de Gualtar
4710-057 – Braga – Portugal
a47408@alunos.uminho.pt

Abstract. Nowadays, there is a lot of technology that must not fail at any circumstances. A car airbag is a simple reactive system that must work every time the car crashes. In order to model problems like this, there are some model-checking applications that can help us do the job. In this paper a well known case study will be modeled and verified in Uppaal and some variants of the problem will be documented. At the end the problem will be generalized in order to make it easy to apply to other problems with different values.

Keywords: Uppaal, timed automata, reactive system, case-study

1 Introduction

This paper presents a case-study on modeling and verification of time-critical systems using timed automata [2] and the Uppaal model checker [4]. The case study revisits a well-known problem on multimedia design: the specification and verification of media streams subject to a number of quality of service constraints, namely end-to-end latency and throughput. The proposed solution improves, in what concerns generality, a previous attempt documented in [5]

The full version of this paper can be found at `http://tinyurl.com/paper55`.

2 Case study: The media stream channel revisited

The problem chosen as a case-study for this paper was a media stream channel. This has three elements: the *Source* that emits messages, the *Sink* that receives and processes them, and the *Channel* that establishes the connection between the *Source* and the *Sink*.

These elements are obliged to the following requirements:

- *Source* emits a message every 50ms;
- *Channel* takes between 80ms and 90ms to deliver the message;
- *Channel* may loose messages, but no more than 20% of them;
- A message is considered lost if it does not arrive at the *Sink* within 90ms;

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 519–522

– *Sink* takes 5ms to process each received message;
– An error should be generated if less than 15 messages per second arrive to the *Sink*.

These requirements should be modeled in Uppaal, first by simulation in order to get a feeling about its behavior and then tested through the verification of suitable queries to check if the envisaged requirements are indeed enforced.

3 Problems and Solutions

One of the problems we have faced was to find a way to avoid the absence of control over the messages. Actually, each message is sent by the *Source* every 50ms, and the *Channel* must have a way to simulate the delay of the communication of that message. Moreover, it must be possible to have more than one message in delay at the same time.

Fortunately, it was realized that two processes alone were enough to solve the problem, because of the *Source* frequency and the *Channel* latency values — this was the crucial observation in this case-study. Such two processes must have the possibility of being reused, inasmuch as when one of them starts processing one message, it will be busy for at most 90ms, and, as in this interval another message may be generated, there must be another process ready to synchronize.

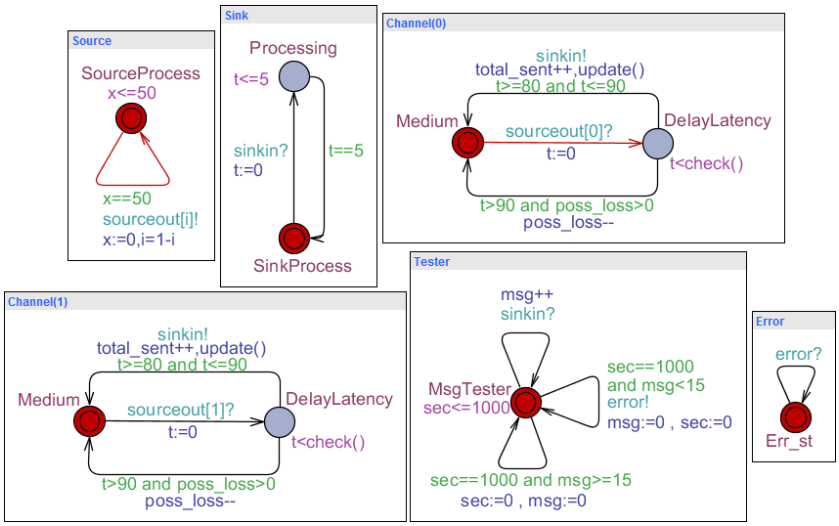

Fig. 1. A possible solution.

In order to make this process reusable, the initial state is always waiting for a message. On its arrival the process resets the clock and goes to a state where it waits for a time t between 80ms and 91ms as shown in Fig. 1. Invariant

$t < check()$ attached to this state caters for the possibility of losing messages. Note that the transitions must have a parameter to identify which *Channel* process will receive the message.

To make sure there is an error if less than 15 messages are processed by the *Sink*, a *Tester* process was created to count the number of successful sinkin synchronizations per second. To do this, the latter increments a variable each time a message is sent and resets the clocks and the counter every seconds, sending an error to the *Error* process if the number of messages is less than 15.

In order to make sure that *Channel* does not loose more than 20% of the messages, the number of total messages and the number of messages lost should be known. But we can't have unbounded integers with this information, because when we try to verify a property, the proof tree would grow till the maximum integer allowed is reached or till the memory runs out. To fix this problem, we resort to an easy calculation: saying that *Channel* may loose messages, but no more than 20% is the same as saying that in every 5 messages, 1 may be lost. So, instead of counting all the messages, we can decide to increment the counter until it reaches 5, and then reset it, incrementing one possibility of loss.

The counter that stores the number of possible losses at any moment still remains in model and must be delimited. Actually it makes no sense for a media channel to have a huge number of possible losses, even because the *Sink* will emit an error if it receives less than 15 messages per second. So it is acceptable if we introduce a little error here, that in theory will decrease the percentage fixed in the requirements, meaning that less losses are allowed. To implement this, the function *update()*, only tolerates the counter *poss_loss* to be incremented until it reaches a MAX value, globally defined. This value will maximize the number of states considered in the proof tree; and the smaller this number, the faster the proof will be.

The check function mentioned above is defined as follows

```
int check()
{ if (poss_loss>0) return 91;
   return 90; }
```

and will enable or disable the transition of loss depending on the value of *poss_loss*.
A property selected for verification was the most obvious:

```
A[] not deadlock
```

i.e. no state reaches a deadlock configuration. Simple as it was, its verification was a considerable challenge, but after the replacement of the unbounded counters, the property was verified.

4 Concluding

The final model, depicted in Fig. 1, can be easily generalized to other media stream channels, with different latencies and frequencies. In order to do that, the frequency of the source , the maximum latency of the channel and the interval of latency should be declared as integer constants respecting the following rules:

```
const int source_freq=10;   //must be greater than Sink process time
const int max_latency=150;
const int latency_interv=5;   //must be <= source_freq
const int N=max_latency/source_freq+1;
```

With the values above, 15 channel processes are created by the Simulator, as the value of constant N is computed to guarantee deadlock avoidance. Restrictions at the source frequency and the latency interval are due to the *Sink* processing time, because there is only one instance of it. Note that the latency interval must be less than the source frequency. Otherwise one process could be trying to send a message to the *Sink* while another message was still being processed. Note that the *Source* must synchronize with more than two processes, so the index of the transition must be incremented until it reaches $N - 1$, being then restarted.

The case study discussed in this paper, set as an exercise on the practical use of Uppaal, illustrated how this sort of tool-supported formal methods can give a precious help to the design of real-time, complex systems. The final solution not only followed a different strategy, but also represents a more general solution to the media stream problem than the one proposed in [5].

The Uppaal simulator is a excellent help when creating a model, because it allows debugging even before the specification is mature and complete enough to be model-checked. Note, however, its role is always limited to that of an animator of the intended behaviour. In this case-study, for example, the verifier found a problem that could not be handled by the the simulator, the latter being unable to pursue exhaustive checking. However, the error found could have been a lot easier to debug if the model-checker had issued a warning, informing that the variable was not defined as a bounded integer and was being incremented.

In concluding, it is a fact that model checking [3] is, at present, a mature collection of techniques and practical tools for automated debugging of complex reactive systems [1]. Actually, as this small case-study may have helped to illustrate, it is no more a theoretical oddity, but, on contrary, its relevance for the working software engineering cannot be understated.

Acknowledgments. I would like to thank to Luís Soares Barbosa for all the support and encouragement he gave me, because without it, I wouldn't have written this short article.

References

1. L. Aceto, A. Ingólfdótir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, specification and verification*. Cambridge University Press, 2007.
2. R. Alur. Timed automata. *Theoretical Computer Science*, 126:183–235, 1999.
3. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
4. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems, 1996.
5. H. Bowman, G. P. Faconti, and M. Massink. Specification and verification of media constraints using upaal. In *Design, Specification and Verification of Interactive Systems'98, Proceedings of the Fifth International Eurographics Workshop, June 3-5, 1998, Abingdon, United Kingdom*, volume 1, pages 261–277. Springer, 1998.